2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)

HybridSFC: Accelerating Service Function Chains with Parallelism

Yang Zhang
University of Minnesota, Twin Cities
yazhang@cs.umn.edu

Zhi-Li Zhang
University of Minnesota, Twin Cities
zhzhang@cs.umn.edu

Bo Han
AT&T Labs – Research
bohan@research.att.com

***Abstract*—Network Function Virtualization (NFV) coupled with Software Defined Networking (SDN) creates new opportunities as well as substantial challenges such as increased Service Function Chain (SFC) latency and reduced throughput. In this paper, we present HybridSFC, a framework that explores the opportunities of parallel packet processing at both traffic level and Network Function (NF) level. It incorporates innovative control and data-plane mechanisms that partition and convert a sequential chain into several (finer-grained) *SFClets* that can be executed *in parallel* on multiple cores and servers. HybridSFC is practical in that it can handle NFs *spanning multiple servers* and requires no modifications to existing NFs. Experiments show that HybridSFC reduces latency up to 51% with 7% CPU overhead, and a 1.42×-1.87× improvement in overall system throughput.**

I. Introduction

Network Function Virtualization (NFV), coupled with Software Defined Networking (SDN), promises to revolutionize networking by allowing network operators to dynamically manage networks. Operators can create, update, remove or scale out/in network functions (NFs) *on demand* [21], [36], [35], construct a sequence of NFs to form an Service Function Chain (SFC) [10], and steer traffic through it to meet service requirements [22], [23], [24]. However, virtualization and "softwarization" of NFs pose many new challenges [11]. In particular, traffic traversing virtualized NFs suffers from reduced throughput and increased latency, compared to physical NFs [13], [22], [23], [24]. The flexibility offered by SDN and NFV enables more complex network services to be deployed, which will likely lead to longer SFC. As the length of an SFC (*i.e.,* number of NFs) increases, so does its overhead.

Exploring parallelism to reduce packet processing latency and increase the overall system throughput is a classical approach that is widely used in networked systems. In terms of NFV, parallelism is first explored in ParaBox [38] and later in NFP [31] by investigating order independence of certain NFs within an SFC. Both efforts focus on parallelizing packet processing for SFCs on a *single* (multi-core) server. Real-world NFs, on the other hand, will likely be operating in edge clouds or data centers with clusters of servers. How to effectively utilize multiple servers to reduce per-packet processing latency and increase the overall system throughput is the main problem we explore.

In this paper, we present HybridSFC, a parallelization framework to accelerate SFCs across multiple servers. HybridSFC converts a sequential SFC into multiple SFClets (i.e., SFC applied over a subset of traffic streams) to better explore *traffic-level parallelism*, and parallelizes NF processing across multiple cores/servers for *NF-level parallelism*. HybridSFC controller calculates optimized paths for each hybrid SFClet [1] and programs both software and hardware switches to enable parallelism across NFs running on multiple physical servers. HybridSFC employs a customized data plane for parallel processing without modifying the implementation of existing NFs. Based on the instructions from controller, HybridSFC data plane *mirrors* packets to parallelizable NFs and then *merges* their outputs. HybridSFC ensures the correctness of SFC processing, *i.e.,* traffic and NF states changed after each SFClet processing are identical to what would have been produced by the original sequential SFC. We make the following contributions:

- We identify the challenges in SFC parallelism (§ II), and design HybridSFC, a framework to support SFC parallelism across multi-core servers (§ III).
- We present HybridSFC controller (§ IV) to enable both NF level and traffic level parallelism. In addition, we present key building blocks in HybridSFC data plane, and explore the placement choices for a high-performance data plane (§ V).
- To demonstrate the effectiveness of HybridSFC (§ VI), we implement a prototype and evaluate it via both synthetic SFCs generated by simulation, and practical SFCs constructed by off-the-shelf open-source and production-grade NFs.

II. Challenges and Related Work

We highlight the key challenges in accelerating SFC processing with parallelism in a cluster of multi-core servers. We also briefly discuss related work.

A. NF Dependence in NF-level Parallelism

The basic premise of SFC *NF-level* parallelism [38], [31] is that given a pair of NFs, if the operations of two NFs applying to the same traffic stream do not conflict, then they can be executed in parallel. For example, if both NFs simply perform read operations, they can be parallelized. If one performs a read operation and another performs a write operation, they can be parallelized if and only if they do not operate on the same header field. Likewise, if both perform write operations (including inserting/removing headers/bits in the packet), they

[1] A hybrid SFClet consists of both parallel segments (with NFs processing in parallel) and sequential segments (with NFs processing sequentially).

cannot be executed in parallel if the modified data portions (header or payload) potentially overlap. Therefore, awareness of NF semantics is required for SFC NF-level parallelism. NF semantics are defined as any operation applied on packets (*e.g.,* modifying packet header/payload, dropping packets, *etc.*) There are two general approaches to infer the semantics of an NF: offline modeling and online monitoring. In offline modeling, the manually created model [8], [14] is known to be error-prone. If NF source code is available, NFactor [34] is able to synthesize NF semantics by performing code refactoring and program slicing. SymNet [30] proposes a symbolic execution friendly language to extract NF semantics. On the other hand, online monitoring (e.g., SIMPLE [27]) adopts similarity-based correlation algorithms to infer the packet modifications applied by an NF through the comparison of its incoming and outgoing packets. Since inferring NF semantics has been extensively explored in the literature, we assume that semantics of each NF in target SFC are available.

B. Traffic-Level Parallelism across Multi-core Servers

Stateful NFs introduce *coupling* among flows for efficient packet processing. In other words, one cannot blindly perform "traffic-level parallelism" by routing packets independently to multiple servers for parallel packet processing and load balancing. Thus, it is important to understand the operator intent [26]. For example, an IDS is configured to detect DoS attacks on a per-host basis. When scaling out NF for scalability and performance, we have to forward traffic generated from the same host to the same NF instance. In a cluster of multi-core servers, there are many factors to consider when deciding *whether* and *how* to parallelize an SFC. For example, it might be beneficial to execute an SFC entirely within a single server, or even within a CPU core – *run-to-completion* [25], [16] to avoid network latency or core switching overheads. However, it will hinge on our ability to load balance the traffic among these servers. The flow coupling of stateful NFs makes this a nontrivial task; in addition, different types of traffic going through the same SFC may incur varying processing overheads; interference among NFs running on the same machine also jeopardizes SFC performance [32]. In contrast, SFC execution can span over multiple servers as *a pipeline*. However, context switching among CPU cores and additional network latency for steering traffic through servers may mitigate the gain in parallel packet processing among multiple servers. Which options are the best in terms of the SFC processing latency and overall system throughputs will depend on individual NF performance, traffic volumes, server and switch capabilities, and network bandwidth. Moreover, *real-world* operational constraints of NF placement (e.g., security concerns requiring some third-party NFs be placed on certain servers) further limit the options of SFC parallelism. Last but not the least, in designing mechanisms for accelerating SFC processing in a multi-server environment, it is important to "co-design" the rules in both hardware and software switches for splitting and steering traffic appropriately. HybridSFC aims at addressing these challenges for accelerating SFC processing.

C. Related Work

SFC optimization has attracted a flurry of research interests in recent years. NFV frameworks such as NetVM [13], BESS [12], [1], and Metron [16] utilize DPDK and smart-NIC/hardware rule offloading to speed up SFC packet processing on commodity servers, whereas OpenBox [7] decomposes vNFs into re-usable modules to further speed up the packet processing pipeline. Flurry [36], NFVnice [21], and E2 [25] exploit flow-level traffic parallelism to improve scalability of NFV. More closely related to our work, ParaBox [38] and NFP [31] utilize parallelism for accelerating an SFC within a single multi-core server. As discussed above, availability of multiple servers not only offers more opportunities for parallelism, but also imposes additional constraints. These issues are not considered in the previous studies, which are the focus and contribution of our paper. The statefulness of NFs is a major hurdle in the SFC optimization that have been studied in many papers [28], [9], [15], [20], [29], [33], [19]. We employ the insights from these studies for converting a SFC into multiple hybrid SFClets to better exploit *both traffic and NF-level parallelism* over multi-cores and across multiple servers. In addition, unlike [37], [31], [25], [7], [16], HybridSFC can accommodate open-source and proprietary NFs with no NF modifications.

III. Architectural Overview

In this section, we present the system architecture and introduce the key components of HybridSFC.

Implementing parallel packet processing is by no means straightforward. First, HybridSFC must guarantee the correctness of generated chain by carefully analyzing the *order dependency* of NFs in a chain. The dependency relies not only on the semantics of NFs, but also their configurations and operational rules. Second, HybridSFC needs to automatically program both virtual and hardware switches by creating appropriate data-plane forwarding rules to perform parallel packet processing across multiple servers. Third, the data plane functions of HybridSFC to support SFC parallelism should be lightweight, avoiding adding too much processing overhead. Finally, to enable incremental deployment, HybridSFC should not require modifications to existing NFs.

HybridSFC consists of a controller and a data plane running on a cluster of multi-core servers equipped with DPDK. The primary role of the controller is three-fold: i) taking each SFC expressed by a network operator, together with the configuration policies and operational rules associated with each NF in the chain, the controller decomposes and refactors it into multiple SFClets to explore traffic-level parallelism; ii) based on order dependency analysis, the controller converts sequential SFClets into hybrid ones to explore NF-level parallelism; and iii) based on the knowledge about NF performance profiles and placement constraints as well as information about server and network resources, the controller schedules the execution of each SFClet to maximize the overall SFClets processing throughput and reduce latency. In contrast, the HybridSFC data plane engine consists of three key building blocks, *traffic steering* (which splits, distributes and also load-balances traffic to appropriate NFs), *mirror* (which duplicates packets for parallel processing), and *merge* (which combines and processes duplicated packets after parallel processing) modules. The primary role of data plane is to execute hybrid SFClets based on the forwarding and processing rules installed by the controller.

IV. HybridSFC Controller

In this section we present the design of the HybridSFC controller which comprises three key modules as shown in Fig. 1, and describe corresponding algorithms.

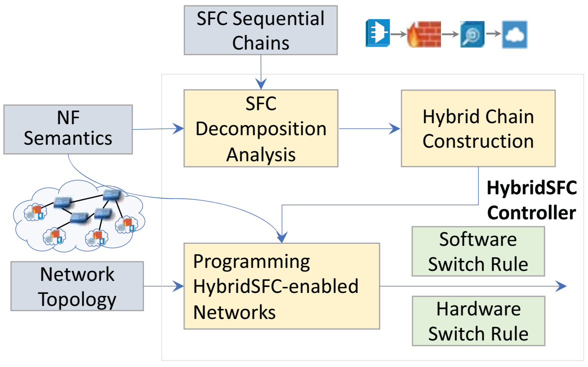


Fig. 1: Overview of HybridSFC controller

A. SFC Decomposition Analysis

Given an SFC – expressed as a network operator or service provider *intent* – consisting of a sequence of vNFs with configuration and policy rules to be deployed, the controller will first invoke the *Service Chain Decomposition* module. This module is to decompose an SFC into multiple "SFClets". As discussed earlier, the challenges lie first in modeling and representing NF semantics, and then synthesizing diverse NF configuration and policy rules. To tackle the challenges, we employ a similar technique used in Header Space Analysis [18], [17] to analyze the configuration and policy rules of each NF (commonly expressed as "match-action" like rules similar to those used in SDN), keep track of the possible header transformations, and synthesize them across the NFs. Based on such analysis, we then decompose the original SFC into a series of SFClets. Our current approach can only synthesize L2-L4 header fields. We will explore NF semantics analysis that goes beyond L4 headers [10] as future work.

B. Hybrid Chain Construction Algorithm

Given SFClets output by decomposition module, hybrid chain construction algorithm examines the semantics (e.g., operations and state variables) of each NF in an SFClet, analyzes the order dependency and determines whether it is safe to execute certain NFs in parallel by converting the sequential SFClet into a hybrid one. We present a heuristic algorithm (Alg. 1) for this analysis. The basic idea is to parallelize two consecutive NFs based on their order dependency constraints,

Algorithm 1 Hybrid Chain Construction Algorithm

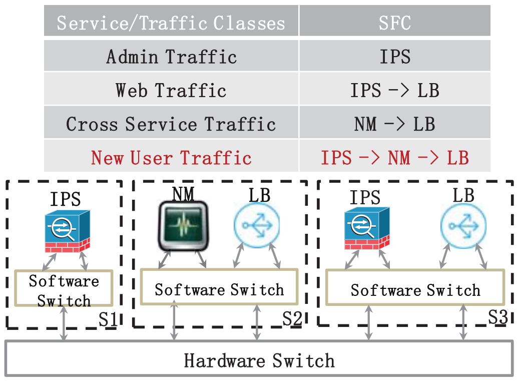
Variable Definition: (a) SC: sequential chain (input); (b) NF_Ops: operations of NF (input); (c) NF_Seg: current NF segment; (d) Agg_Ops: aggregated operations of current segment; (e) HC: hybrid chain (output)

```
1:  procedure CONSTRUCT_HYBRID_CHAIN
2:      initiate $NF\_Seg, Agg\_Ops, HC$
3:      while $NF_i$ in $SC$ do
4:          $NF\_Ops \leftarrow Fetch\_Ops(NF_i)$
5:          if $Independent(NF\_Ops,\ Agg\_Ops)$ then
6:              $NF\_Seg.push(NF_i)$
7:              $Agg\_Ops.push(NF\_Ops)$
8:          else
9:              $HC.push(NF\_Seg)$
10:             $NF\_Seg.clear(); Agg\_Ops.clear()$
11:             $NF\_Seg.push(NF_i)$
12:             $Agg\_Ops.push(NF\_Ops)$
13:     $HC.push(NF\_Seg)$
```

aggregate their operations, and take their combination as a "bigger" NF for further processing. If NF_i is parallelizable with the current NF segment NF_Seg (*i.e.,* having independent ordering which is derived from their operations), we push it into NF_Seg and aggregate its operations into Agg_Ops (lines 6-7). Otherwise, we push NF_Seg into the output hybrid chain HC as a completed segment, clear NF_Seg and Agg_Ops, and then push NF_i into NF_Seg and its operations into Agg_Ops for the order-dependency check with the next NF (lines 9-12).

C. Programming HybridSFC-enabled Networks

Given a hybrid SFClet, HybridSFC controller takes the performance profiles of each NF, server, network resource profiles and constraints (NF placement policy considerations) to decide its execution, weighing various options by considering performance benefits and service level objectives/agreements.



Service/Traffic Classes	SFC
Admin Traffic	IPS
Web Traffic	IPS -> LB
Cross Service Traffic	NM -> LB
New User Traffic	IPS -> NM -> LB

Fig. 2: Traffic Distribution Example.

Consider the example shown in Fig. 2 where the placement of NF instances has already been determined. Given a new SFC, IPS $\rightarrow$ Network Monitoring (NM) $\rightarrow$ LB, where NM and

LB can be executed in parallel. This yields multiple options for traffic distribution and steering between the NFs with possibly different network bandwidth and latency implications: 1) $IPS_{1,3} \rightarrow NM_2 \rightarrow LB_{2,3}$; 2) $IPS_{1,3} \rightarrow LB_{2,3} \rightarrow NM_2$; 3) $IPS_{1,3} \rightarrow NM_2 || LB_{2,3}$, where we use $||$ to denote parallel execution, and the subscripts are server id.

Our SFClet execution algorithm consists of four steps: 1) extract common NF subsequence between target SFClet and NF instances running on each server (*e.g.,* $< IPS >_1$, $< NM \rightarrow LB >_2$, $< IPS \rightarrow LB >_3$, *etc.*); 2) compute possible path combinations that meet the target SFClet processing requirements (*e.g.,* $IPS_1NM_2LB_3$, $IPS_3NM_2LB_3$, $IPS_1NM_2LB_2$, *etc.*); 3) find the paths with minimal number of servers (*e.g.,* $IPS_3NM_2LB_3$, $IPS_1NM_2LB_2$, *etc.*); 4) select the paths with the most parallelizable NF segments (*e.g.,* $IPS_1NM_2LB_2$, *etc.*) on the same servers. The intuition behind this algorithm is that inter-processor/core communication within a single server is faster than across multiple servers, and parallelized NF execution reduces processing latency.

We remark that one SFClet execution solution that yields better performance at the current moment may no longer be the case later when the traffic load increases or system resources change (e.g., due to failures). We leave it as a future work to adaptively adjust traffic distribution and dynamically scale out/in NF instances in response to changes in system resources and NF/server/network failures.

V. HybridSFC Data Plane

We present the data-plane design of HybridSFC, focusing on the mirror and merge modules and their placement.

A. Mirror and Merge Modules

We show the main building blocks and data structures of the HybridSFC data plane in Figure 3. The data-plane execution engine enforces SFC parallelism based on the forwarding and processing rules installed by controller. We first present the design of data plane within a single server, and then present the multiple server case.

Before presenting the details of mirror and merge modules, we first describe two supporting tables, *Flow Steering* table and *Packet State* table. The Flow Steering table contains information for packet processing of SFC segments consisting of NFs residing in the server. Each entry represents an SFC segment, *e.g.,* {A,{B,C}}, along with the corresponding NF operations denoted as OPS (see § IV-B), and FID which shows the flow (in terms of a layer 2–4 header match rule) to which the SFC segment applies. Controller installs these entries in the software switch, and then mirroring module uses the Flow Steering entries to steer packets along NFs, duplicating them if needed. For an example SFC segment {A,{B,C}}, mirroring module duplicates packets processed by A, and then sends them to both B and C for parallel processing.

The Packet State table is primarily used by the merge module and contains four fields: 1) PID (packet ID), 2) PKTR (reference pointer to the memory address of the original packet), 3) BUF (packet buffer for saving the intermediate

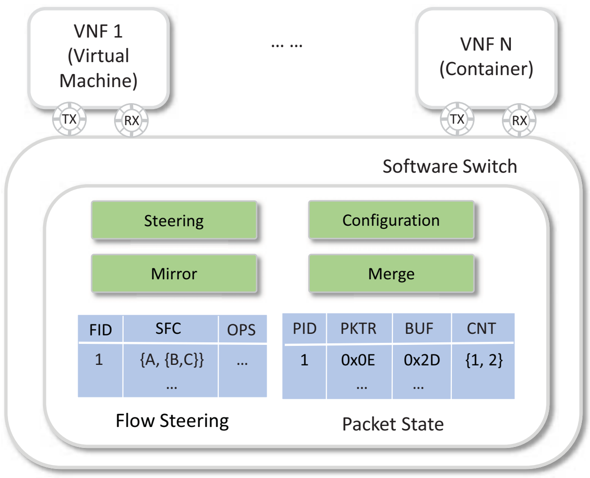


Fig. 3: Overview of HybridSFC data plane

results), and 4) CNT (counter array for parallel SFC segments). The unique PID keeps track of packets that are processed by parallelized NFs. The CNT records the number of parallelized NFs in each segment. For instance, CNT for {A, {B, C}} is {1, 2}. The count decrements by one after a packet processed by a parallelized NF. Merge operation is triggered when the count reaches zero.

In the merge operation, we treat a data packet as a sequence of bits, namely a $\{0|1\}^*$ string. If an NF in a processing segment adds L extra bits into packets, we insert a string of L zeros at the corresponding location in the outputs from other NFs before the merge. In a similar vein, if an NF removes L bits from packets, we delete the same bits from the outputs of other NFs. Moreover, if packets are dropped, a no-op packet is sent back to merge module.

Assume P_O is the original packet and there are two NFs A and B in the chain. Assume that P_A and P_B are respective outputs from NF A and B. The packet output after two NFs will be $P_O \oplus P_A \oplus P_B$. Note that correctness is guaranteed as two NFs do not modify the same portion of a packet. Finally, we recalculate the checksum before steering/mirroring the packet to the next NF(s) – either residing within the same server or different servers.

B. Placement of Mirror and Merge

When parallelizing packet processing for NFs spanning multiple machines, a natural question is where to place mirror and merge functions. For SFC parallelism within a server, mirror and merge functions are placed in software switches, as in ParaBox [38], or merge function can run as a dedicated container in NFP [31]. However, when considering an SFC spanning multiple servers, if we naively place mirror and merge modules on arbitrary servers, we may not only waste bandwidth, but also potentially increase the SFC latency.

For example, consider an SFC: NAT $\rightarrow$ FW $\rightarrow$ IPS $\rightarrow$ WANX, and suppose we can parallelize the packet processing for NAT, FW and IPS, but not WANX. Further, assume that NAT and FW are placed on server 1, while IPS and WANX are on server 2. If we place the mirror and merge functions on server 1, the mirror function has to duplicate packets to IPS on server 2. After IPS examines the packets, it has to send them back to server 1 for the merge, and then back to server 2 again for the last-hop WANX processing. As a result, placing mirror and merge functions on server 1 ends up increasing latency. Instead if we place mirror and merge intelligently, *i.e.,* mirror on server 1 and merge on server 2, then we can reduce the extra traversal of packets caused by placing both mirror and merge on same server.

Hence, when parallelizing SFC processing across multiple servers (with NF placement constraints), the placement of mirror and merge functions is crucial. Using two parallelized NFs placed on two servers (one on each) as an example, below are several design choices for the placement. (a) *decoupling mirror and merge into different software switches*: this placement strategy can, to certain extent, avoid sending packets back-and-forth between the two servers, but there will still be two outgoing flows from server 1 and two incoming flows into server 2 (in contrast, processing two NFs sequentially generates only one incoming and outgoing flow at each server). (b) *flexibility of putting mirror on hardware switches and merge on software switches*: this placement can reduce the number of outgoing flows from server 1; it still cannot improve the situations (two incoming flows) at server 2. This design choice is what has been currently implemented in HybridSFC. (c) *flexibility of putting both mirror and merge on hardware switches*: this is the ideal case to achieve reduced latency with the same bandwidth utilization as the sequential chain.

Although it is feasible to place the mirror function on hardware switches, it is challenging to design and implement merge function even on programmable hardware switches. The reason is that the merge function requires relatively complex logic and needs extra memory to store intermediate results.

VI. Evaluation

We evaluate HybridSFC through a prototype implementation, and show results for the following scenarios.

- In benchmarking experiments, HybridSFC performance is better compared to existing solutions (Figure 5).
- In realistic chains, HybridSFC reduces SFC latency in various setups (Figure 6).
- HybridSFC improves packet processing performance in multi-server scenarios (Figure 7, Figure 8).
- The overhead introduced by HybridSFC is manageable (Figure 9, Figure 10).

Experimental Setup. The prototype uses an Openflow-enabled switch with 48 10Gbps ports. We connect six servers to the switch, four of which are equipped with two 10Gbps links, and the other two are equipped with four 10Gbps links. Each server uses Intel(R) Xeon(R) CPU E5-2620 with 6 cores. On each server, one core is dedicated to HybridSFC data plane. The HybridSFC controller runs on a standalone server that is connected to the management port of the switch on a separate 1Gbps control network.

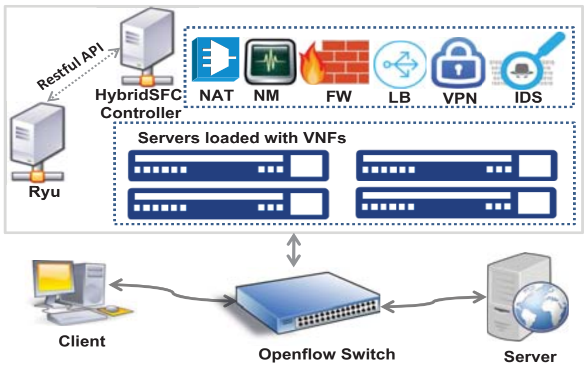


Fig. 4: Experiment Setup

Berkeley Extensible Software Switch (BESS) [1] is used for implementing data plane logic. Regarding the control plane, we have implemented a controller and a local daemon in Python using Flask framework. Table entries are stored in pickleDB which is a light-weight key-value store. The HybridSFC controller communicates with a Ryu OpenFlow controller through its RESTful APIs. Packet size in experiments is 64B, and Pktgen-dpdk [6] is used as traffic generator. **NFs in Experiments.** Each NF is running in either a Docker container or a KVM-based VM. We dedicate a CPU core to each container or VM. We use seven types of NFs: Layer 2 forwarder (L2FWD), NAT, FW, IDS, Monitor, Load Balancer, and VPN gateway. L2FWD is used for NF benchmarking. For NAT and FW running in VM, we use product-level NFs with real operational rules from a carrier network. We also use open-source iptables running in containers as NAT and FW. We use BRO [2] as IDS, Nload [4] as Monitor, Linux Network Load Balancing [3] as Load Balancer, and OpenVPN [5] as VPN gateway in the experiments. Moreover, we create customized L2FWD, NAT, and FW to invoke BESS zero-copy API and OpenNetVM zero-copy API respectively for benchmarking purpose.

A. HybridSFC Performance

As a benchmarking experiment, Figure 5 shows the latency comparison among sequential SFC, hybrid SFC by HybridSFC, and OpenNetVM [37]. L2FWD is used as NF instance, and SFC length is increased from 1 to 6. We observe that the processing latency in parallel processing is significantly reduced compared to sequential processing, and both the sequential and parallel chains perform better than OpenNetVM. Moreover, the latency reduction rises from 13.04% to 50.98% as SFC length increases.

Table I presents sequential and corresponding hybrid chains generated. NFs in () mean that they can be processed in parallel, and NFs separated by || mean that they are on different servers. HybridSFC can reduce the SFC latency by up to 31.7% as shown in Figure 6.

TABLE I: Service Function Chains Used in the Experiments

Index	Deployed Chain in One Machine	Hybrid Chain in One Machine	Deployed Chain in Two Machines	Hybrid Chain in Two Machines
1	{IDS, NAT, FW}_container	{(IDS, NAT, FW)}	{IDS ‖ NAT, FW}_container	{(IDS, ‖ NAT, FW)}
2	{IDS, NAT, FW}_vm	{(IDS, NAT, FW)}	{IDS ‖ NAT, FW}_vm	{(IDS, ‖ NAT, FW)}
3	{VPN, Monitor}_vm, {FW, LB}_container	{(VPN), (Monitor, FW), (LB)}	{VPN, Monitor}_vm ‖ {FW, LB}_container	{(VPN), (Monitor ‖ FW), (LB)}
4	{NAT, FW, IDS, LB}_vm	{(NAT, FW, IDS), (LB)}	{NAT, FW ‖ IDS, LB}_vm	{(NAT, FW ‖ IDS), (LB)}

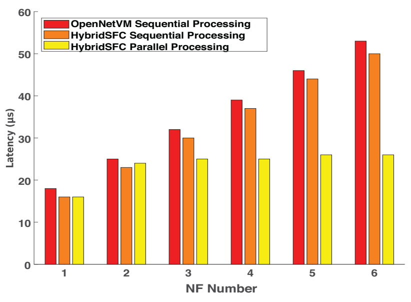


Fig. 5: Performance benchmarking results

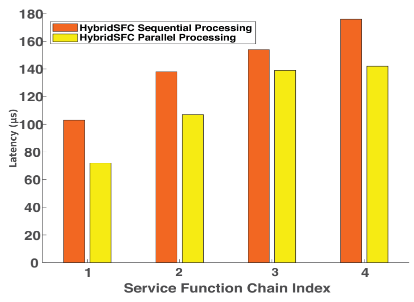


Fig. 6: Performance of various SFCs with different complexity

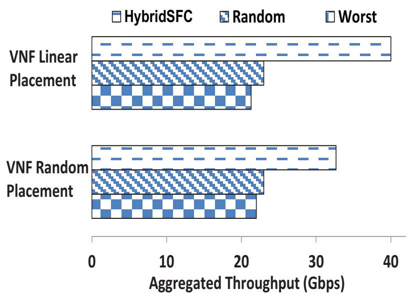


Fig. 7: Throughput in different placement solutions

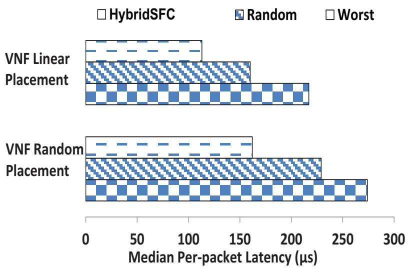


Fig. 8: Latency in different placement solutions

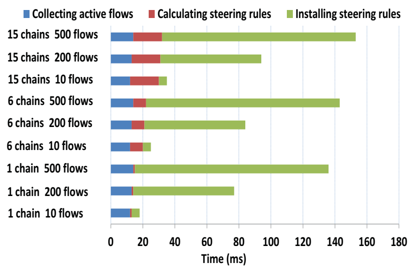


Fig. 9: Breakdown of controller processing time

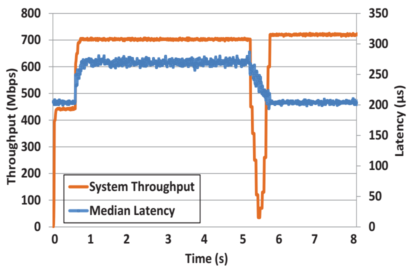


Fig. 10: Overhead during NF Scaling

Next, we investigate the impact of NF placement spanning over multiple servers. In linear placement, three types of zero-copy NFs (L2FWD, NAT, FW) are loaded into each of the four servers (12 instances in total); in random placement, the 12 NF instances are placed randomly. We configure traffic to go through four SFCs (L2FWD→NAT→FW, L2FWD→NAT, L2FWD→FW, NAT→FW). 500 flows in each SFC are generated. We present the results of HybridSFC, random distribution approach, and the worst case. The aggregated throughput is shown in Figure 7, and latency result is shown in Figure 8. HybridSFC achieves 1.74x-1.87x throughput improvement in NF linear placement while 1.42x-1.48x throughput improvement in NF random placement. Similarly, we can also observe latency benefits of HybridSFC compared with others.

In order to verify the correctness of HybridSFC, we always send traffic through sequential SFCs, replay the same traffic in HybridSFC, and then compare the packets at receiver side and network states maintained in NFs using log information.

B. HybridSFC Overhead

To understand the overhead introduced by HybridSFC, especially the data plane logic, we measure the CPU cycles of processing each packet in benchmarking experiments which uses L2FWD as NF. CPU cycles per packet are increased with the SFC length in both sequential and hybrid chains. However, CPU cycles per packet in HybridSFC parallel processing is up to 7% more than the sequential SFCs.

The HybridSFC controller needs to calculate appropriate rules and install into hardware and software switches. Thus, controller overhead is segmented into three parts: 1) collecting active flows; 2) calculating steering rules; 3) installing steering rules. We create 9 scenarios and manually trigger topology update which forces the controller to recalculate traffic distribution rules. We measure each overhead segment, as shown in Figure 9. We observe that the overhead of calculating steering rule depends on the number of chains in SFC configuration, while the overhead of installing steering rules depends on the number of active flows in existing network. Since controller only recalculates steering rules when topology is changed, the costly operation is affordable.

During NF scaling, we want to further investigate the overhead and observe how throughput and latency change. We deploy two servers, one of which only runs an IDS, while the other one runs an IDS and a L2FWD. We then generate traffic going through IDS → L2FWD. As shown in Figure 10, the throughput of the system increases immediately and reaches the bottleneck of IDS processing. After that, HybridSFC controller gets notification from local daemon running on the NF server and uses a sub-optimal path (utilizing

IDS on the other machine too). The system throughput goes up because two IDS instances are utilized, but median latency increases as well because certain traffic is detoured into a sub-optimal path which spans over two servers. At the fifth second, we spin up another L2FWD in the server running only one IDS, and trigger the controller to recalculate the traffic distribution rules. In our current implementation, after controller recalculates traffic distribution rules, it will refresh existing rules. This will cause temporary system unavailability, and thus system throughput drops down rapidly, but recovers quickly. This is a trade-off of NF performing scaling without NF modification, and we have been exploring an effective mechanism of conquering it. Now both SFClets are optimized, so latency drops.

VII. Conclusions

In this paper, we have presented HybridSFC, a parallelization framework to accelerate SFC across multi-core servers. In contrast to existing approaches, HybridSFC performs both NF and traffic level parallelism without NF modification. HybridSFC takes into account the configurations and operational rules of NFs to create more parallelization opportunities. HybridSFC supports not only NFs within a single server, but also those spanning multiple machines, by distributing traffic over optimized SFCs. The data plane of HybridSFC runs as an extension of BESS to achieve high performance and programmability. Our evaluation demonstrates that HybridSFC can significantly improve SFC performance with manageable overheads under various realistic scenarios.

VIII. Acknowledgment

The research was supported in part by NSF under Grants CNS-1618339, CNS-1617729, CNS-1814322, and CNS-1836772.

References

[1] Berkeley Extensible Software Switch. http://span.cs.berkeley.edu/bess.html, 2019.

[2] BRO: Network Intrusion Detection System. https://www.bro.org/, 2019.

[3] Linux Network Load Balancing. http://lnlb.sourceforge.net/, 2019.

[4] Network Monitor. https://linux.die.net/man/1/nload, 2019.

[5] OpenVPN: VPN Gateway. https://openvpn.net/, 2019.

[6] Pktgen DPDK. http://pktgen-dpdk.readthedocs.io/en/latest/, 2019.

[7] Bremler-Barr, A., Harchol, Y., and Hay, D. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *Proc. of SIGCOMM* (2016).

[8] Fayaz, S. K., Yu, T., Tobioka, Y., Chaki, S., and Sekar, V. Buzz: Testing context-dependent policies in stateful networks. In *Proc. of NSDI* (2016).

[9] Gember-Jacobson, A., Viswanathan, R., Prakash, C., Grandl, R., Khalid, J., Das, S., and Akella, A. OpenNF: Enabling Innovation in Network Function Control. In *Proc. of SIGCOMM* (2014).

[10] Halpern, J. M., and Pignataro, C. Service Function Chaining (SFC) Architecture. https://tools.ietf.org/html/rfc7665, 2015.

[11] Han, B., Gopalakrishnan, V., Ji, L., and Lee, S. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine* (2015).

[12] Han, S., Jang, K., Panda, A., Palkar, S., Han, D., and Ratnasamy, S. SoftNIC: A Software NIC to Augment Hardware. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html, 2015.

[13] Hwang, J., Ramakrishnan, K. K., and Wood, T. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *Proc. of NSDI* (2014).

[14] Joseph, D., and Stoica, I. Modeling middleboxes. *IEEE Network: The Magazine of Global Internetworking* (2008).

[15] Kablan, M., Alsudais, A., Keller, E., and Le, F. Stateless network functions: Breaking the tight coupling of state and processing. In *Proc. NSDI* (2017).

[16] Katsikas, G. P., Barbette, T., Kostić, D., Steinert, R., and Jr., G. Q. M. Metron: NFV service chains at the true speed of the underlying hardware. In *Proc. NSDI* (2018).

[17] Kazemian, P., Chang, M., Zeng, H., Varghese, G., McKeown, N., and Whyte, S. Real Time Network Policy Checking using Header Space Analysis. In *Proc. of NSDI* (2013).

[18] Kazemian, P., Varghese, G., and McKeown, N. Header space analysis: Static checking for networks. In *Proc. of NSDI* (2012).

[19] Khalid, J., and Akella, A. Correctness and performance for stateful chained network functions. In *Proc. of NSDI* (2019).

[20] Khalid, J., Gember-jacobson, A., Michael, R., Abhashkumar, A., and Nsdi, I. Paving the Way for NFV : Simplifying Middlebox Modifications Using StateAlyzr. In *Proc. of NSDI* (2016).

[21] Kulkarni, S. G., Zhang, W., Hwang, J., Rajagopalan, S., Ramakrishnan, K., Wood, T., Arumaithurai, M., and Fu, X. NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains. In *Proc. of SIGCOMM* (2017).

[22] Kumar, S. Service Function Chaining Use Cases In Data Centers. https://tools.ietf.org/html/draft-ietf-sfc-dc-use-cases-06, 2016.

[23] Nadeau, T., and Quinn, P. Problem Statement for Service Function Chaining. RFC 7498, 2015.

[24] Napper, J. Service Function Chaining Use Cases in Mobile Networks. https://tools.ietf.org/html/draft-ietf-sfc-use-case-mobility-07, 2016.

[25] Palkar, S., Lan, C., Han, S., Jang, K., Panda, A., Ratnasamy, S., Rizzo, L., and Shenker, S. E2: A Framework for NFV Applications. In *Proc. of SOSP* (2015).

[26] Prakash, C., Lee, J., Turner, Y., Kang, J.-M., Akella, A., Banerjee, S., Clark, C., Ma, Y., Sharma, P., and Zhang, Y. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. *Proc. of SIGCOMM* (2015).

[27] Qazi, Z. A., Tu, C.-C., Chiang, L., Miao, R., Sekar, V., and Yu, M. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proc. of SIGCOMM* (2013).

[28] Rajagopalan, S., Williams, D., Jamjoom, H., and Warfield, A. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proc. of NSDI* (2013).

[29] Sherry, J., Gao, P. X., Basu, S., Panda, A., Krishnamurthy, A., Maciocco, C., Manesh, M., Martins, J. a., Ratnasamy, S., Rizzo, L., and Shenker, S. Rollback-recovery for middleboxes. In *Proc. of SIGCOMM* (2015).

[30] Stoenescu, R., Popovici, M., Negreanu, L., and Raiciu, C. Symnet: Scalable symbolic execution for modern networks. In *Proc. of SIGCOMM* (2016).

[31] Sun, C., Bi, J., Zheng, Z., Yu, H., and Hu, H. NFP: Enabling Network Function Parallelism in NFV. In *Proc. SIGCOMM* (2017).

[32] Tootoonchian, A., Panda, A., Lan, C., Walls, M., Argyraki, K., Ratnasamy, S., and Shenker, S. Resq: Enabling slos in network function virtualization. In *Proc. of NSDI* (2018).

[33] Woo, S., Sherry, J., Han, S., Moon, S., Ratnasamy, S., and Shenker, S. Elastic scaling of stateful network functions. In *Proc. of NSDI* (2018).

[34] Wu, W., Zhang, Y., and Banerjee, S. Automatic synthesis of nf models by program analysis. In *Proc. of HotNets* (2016).

[35] Zave, P., Ferreira, R. A., Zou, X. K., Morimoto, M., and Rexford, J. Dynamic Service Chaining with Dysco. In *Proc. of SIGCOMM* (2017).

[36] Zhang, W., Hwang, J., Rajagopalan, S., Ramakrishnan, K., and Wood, T. Flurries: Countless fine-grained nfs for flexible per-flow customization. In *Proc. of CoNEXT* (2016).

[37] Zhang, W., Liu, G., Zhang, W., Shah, N., Lopreiato, P., Todeschi, G., Ramakrishnan, K., and Wood, T. OpenNetVM: A Platform for High Performance Network Service Chains. In *Proc. of HotMIddlebox* (2016).

[38] Zhang, Y., Anwer, B., Gopalakrishnan, V., Han, B., Reich, J., Shaikh, A., and Zhang, Z.-L. ParaBox: Exploiting Parallelism for Virtual Network Functions in Service Chaining. In *Proc. of SOSR* (2017).